Incremental Physical Resynthesis for Timing Optimization

Peter Suaris, Lungtien Liu, Yuzheng Ding, Nanchi Chou
Mentor Graphics Corporation, 8005 Boeckman Road, Wilsonville, OR 97070
{peter_suaris, lungtien_liu, eugene_ding, nanchi_chou}@mentor.com

ABSTRACT

This paper presents a new approach to timing optimization for FPGA designs, namely *incremental physical resynthesis*, to answer the challenge of effectively integrating logic and physical optimizations without incurring unmanageable runtime complexity. Unlike previous approaches to this problem which limit the types of operations and/or architectural features, we take advantage of many architectural characteristics of modern FPGA devices, and utilize many types of optimizations including *cell repacking*, *signal rerouting*, *resource retargeting*, *and logic restructuring*, accompanied by efficient incremental placement, to gradually transform a design via a series of localized logic and physical optimizations that verifiably improve overall compliance with timing constraints. This procedure works well on small and large designs, and can be administered through either an automatic optimizer, or an interactive user interface. Our preliminary experiments showed that this approach is very effective in fixing or reducing timing violations that cannot be reduced by other optimization techniques: For a set of test cases to which this is applicable, the worst timing violation is reduced by an average of 42.8%.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids – *Automatic synthesis, Optimization;* B.7.1 [**Integrated Circuits**]: Types and Design Styles – *Gate arrays;* B.7.2 [**Integrated Circuits**]: Design Aids – *Layout, Placement and routing;* B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids.

General Terms

Algorithms, Performance, Design.

Keywords

FPGA, Logic synthesis, Placement, Timing Optimization

1. INTRODUCTION

Timing optimization for FPGA based designs has always been an important aspect, especially because compared with ASIC technology, FPGA technology has a performance disadvantage due to the flexibility it offers. Evolution of FPGA hardware has been addressing this issue, both through improved process technology (the use of deep sub-micron technologies) and through architecture enhancements (multi-level, flexible logic resources with fast internal interconnects and multiple configurations; dedicated resources for special needs, from memory and multiplier to custom logic cores; and active routing resources, to name a few). Such improvements have enabled designs running at clock frequencies of hundreds of megahertz to be successfully implemented on FPGA devices, and encouraged more high performance designs to target FPGA technology.

Logic optimization (logic synthesis and technology mapping) plays an important role in producing high performance FPGA based designs, as it shapes the logic structure of the designs. However, properly modeling timing constraints for logic optimization has always been a challenge, since such constraints are expressed with respect to the finished implementation, where the logically optimized netlist has been embedded onto the pre-fabricated FPGA device - packed and placed into cells at fixed locations and connected with existing, though programmable, interconnects. Such an embedding process (*physical optimization*, generally including placement and routing) introduces delay variations that are not always well correlated to the netlist structure, and are difficult to estimate a priori.

In the early days of FPGA technology (when architectures were simpler and devices were slower), delays of logic cells were dominant, and netlist structure based delay modeling worked well. Many studies were done using such simplified device and delay models (see [5] for a review of many algorithms), resulting in a family of elegant algorithms, some achieving provable optimality under certain conditions [2]. Some of these are still widely used in today's FPGA logic optimization procedures.

As the architecture becomes more complicated and interconnect delays become dominant due to the use of deep sub-micron technologies, such models become less reliable. To facilitate effective logic optimization, interests have risen on *physical synthesis*, where logic optimization is performed simultaneously or iteratively with physical optimization, so that each part can gain more accurate information from, and pass more direct instructions to, the other part, yielding a better-coordinated effort.

This paper presents a new approach to physical synthesis, which we call *incremental physical resynthesis* (*IPR*). It works on a completed FPGA design that has unmet timing constraints. Incrementally, it identifies small critical sections of the design, and based on the logic and physical structure of the section, timing requirement, and available logic resources, it tries a set of logic and physical optimizations in order to improve timing. Each optimization is a fully integrated logic and physical operation, bringing logically resynthesized and physically valid changes to the design, with verified performance improvement. Being incremental, the time complexity of this approach is manageable even for the larger designs; moreover, it can be controlled by either an automatic optimization driver, or a user interface where the designer can select the spot to optimize and get immediate feedback. Our preliminary experimental results showed that this is an effective tool to fix timing (constraint) violations that have not been fixed by the general optimization procedures including global physical synthesis: on a set of test cases the average amount of reduction on timing violations is 42.8%.

The rest of this paper is organized as follows. Section 2 reviews previous works and presents our motivation. Section 3 describes the overall IPR approach, followed by detailed discussion of optimization operations in Section 4. Section 5 discusses some experimental results; Section 6 concludes the paper.

2. Previous Work and Motivation

Integrating logic and physical optimizations (*physical synthesis*) is a challenging task, in that both sub-tasks are by themselves complicated and time-consuming, and putting them together would significantly increase the complexity. To deal with this difficulty, at least two approaches can be used. One is to limit the types of interaction between the two and/or to limit the types of optimizations involved (which we refer to as the "*reduced approach*"); the other is to limit the scope of the optimizations (which we refer to as the "*incremental approach*"), by incrementally working on small sections. In either case, the solution space that is being explored is limited. Therefore, physical synthesis will likely be a restricted version from the perspective of logic optimization, and serves as an improvement, rather than replacement, of standard logic optimization.

A number of studies on FPGA timing optimization via physical synthesis have been published recently [12,16,17,18], concerning both sequential and combinational logic optimizations.

Sequential logic optimization relies on accurate delay information in order to balance the delay of the paths; physical synthesis has a clear advantage here. In [17], retiming optimization is integrated with placement in two ways. Firstly, cost function of the placement algorithm is modified to identify paths that are hard to improve through retiming, so extra effort can be made to optimize them during placement. Secondly, the standard retiming algorithm is modified to apply on a placed netlist, while limiting disturbance to the placement, by avoiding certain situations that will require more placement changes, and by using an incremental re-placement algorithm based on overlap removal, to minimize placement changes when change is necessary. In [18], an alternative physical retiming algorithm is proposed, where the register movement is constrained by a "budget" calculated using layout based delay information. It allows fast, concurrent retiming of many paths and is effective in practice. This work also includes local logic transformations to enable otherwise un-retimable logic to be retimed.

Combinational logic synthesis for timing optimization involves reduction of path delays, thus also benefits from physical delay information. In [16], logic duplication is performed after placement. With the help of a modified packing routine that strategically leaves logic resources for future logic duplication, creation of new logic via duplication would not disrupt the placement. In [12], simultaneous technology mapping and placement is proposed. Given an initial k-LUT mapping and placement, the netlist is decomposed in-site using gate decomposition, then re-mapped for delay minimization, using layout based delay calculation as guide. Each newly mapped k-LUT is placed into the site where the "root" gate of the LUT used to reside. This is followed by a separate placement phase for overlap removal and improvement.

These are global optimization algorithms, and generally follow the "reduced approach" above, by limiting the integrated optimizations to certain operations (for combinational logic optimization, only node duplication, gate decomposition, and clustering were used in the above works). They also have one notable common objective, which is to keep the physical changes minimal, by trying to avoid major disruption to existing placement, and by tailoring the logic changes towards this objective. Maintaining stability of overall physical structure is a very useful strategy in physical synthesis when physical timing information is not immediately updated during logic optimizations, since substantial changes in physical layout may invalidate the currently used timing information, obtained based on previous physical layout. In this aspect, these works are fundamentally better compared to a direct iteration of logic and physical optimizations, which may run into convergence problems.

While these algorithms are elegant and shown effective for purposed applications, their power can be limited by the restriction on the types of optimizations involved. Moreover, they also used simplified architecture models. Adopting a more sophisticated set of optimizations and applying on real world architectures for large designs will substantially increase the complexity of the algorithms. On the other hand, the "incremental approach", if proven useful, can be more intensive and versatile because it is more focused. These two approaches can compliment and improve each other to produce better result together. This is our motivation of proposing the *incremental physical resynthesis* (*IPR*) approach.

In arguing the usefulness of IPR, we consider that in a typical design flow, a design that is subject to IPR has been processed by standard logic and physical optimizations, and possibly also through global physical synthesis (most noticeably retiming and replication [18]). On such designs, there can be several types of opportunities unique to IPR.

- Designs that have reached this stage usually only have relatively few places with timing violations - if a design has massive violations, it is usually necessary to revisit the design at a higher level, restructure the constraints, and/or reconsider the choice of device. So our objective is to "fix" the problems; being thorough and nimble, IPR can suite the job well.
- Large designs are usually hierarchical, and due to run-time and other concerns, are usually optimized hierarchically during logic synthesis. While the modules may have been intensively optimized, the inherent redundancy of a module design can still be revealed when the design enters the physical domain, where it is flattened onto the device, and cross-hierarchy logic optimizations are possible. Such opportunities are generally around the hierarchical boundaries. Trying to resynthesize the entire flattened netlist is too costly and a wasted effort to most parts of the design, while the more focused IPR would fare better.
- The logic cells of a modern FPGA are very versatile, and different configurations have different logic capacity as well as different timing characteristics. It is difficult for logic synthesis and technology mapping to consider all possibilities, and their consequences, when dealing with a

large design and without knowledge about physical layout. Therefore, a simplified approach is common place; for example, LUT-centric logic synthesis is typical for SRAM based FPGAs. When a small section of the design is concerned and with accurate information on resource availability/distribution and timing characteristics, IPR can do a more thorough search in solution space, and better utilize the device features.

- The dedicated resources for specific logic (such as memories) are very efficient instruments with higher logic density and performance, but their limited present on a device may occasionally cause layout difficulty that can nullify or even reverse their advantage. In general, it is best for logic synthesis to use them as much as possible when their whereabouts are not clear, and for physical synthesis to correct the improperly used ones. These are local transformations and fit IPR well. Similarly, unconventional use of such resources (e.g. using dedicated memory cells for logic [8]) is better determined during IPR when resource availability and distribution are known.

Our IPR approach tries to exploit these opportunities. In addition, we integrate logic and physical transformations at operational level, thus is able to alter placement as needed, without worrying about inaccurate timing information or convergence. To allow efficient update of timing information, which is needed for the transformations, we use only combinational logic optimization operations, without altering the state diagram of the design. This makes incremental timing analysis possible. Note that this is not a real limitation, since sequential optimizations such as retiming [17,18] are inherently global, and can be performed prior to IPR.

3. Incremental Physical Resynthesis

In this section, we outline the procedure of our incremental physical resynthesis (IPR) algorithm. Throughout this and the next sections we will be using the Xilinx Virtex II architecture [19] as the platform for presentation. The overall approach applies to other modern FPGA architectures as well; however, since IPR is tuned to utilize architectural features, the specific optimization operations will have to be adapted to and/or modified for each family of FPGA devices.

Briefly, IPR consists of a number of iterations of localized optimizations. Each round is usually preceded with physical timing analysis to reveal where the optimization effort should be centered. After that, an expandable sub-circuit, which will be the focus of the optimization, is identified. Depending on the characteristics of the sub-circuit, a number of logic transformations may be attempted on it. Each successful and potentially beneficial transformation (in terms of timing improvement) is then incrementally placed, and its impact on physical timing is analyzed. If the benefit is realized through placement, the transformation is accepted, and this round of optimization is completed; otherwise it is reversed, the design is restored to previous state, and this round is also completed.

As stated earlier, we assume that in a design flow IPR is positioned after regular logic and physical optimizations (synthesis, mapping, placement and routing) and optionally global physical synthesis (physical retiming and replication [18], etc.). Although IPR can incorporate a complete physical optimization component that performs both placement and routing for each logic transformation, we have only built an incremental placer; the justification and consequence will be discussed shortly.

3.1 Physical Timing Analysis

As is for any timing optimization algorithm, timing analysis performs a crucial role for IPR. The basic purpose of IPR is to eliminate or reduce violations of timing constraints, a.k.a. *timing violations*, which are exemplified by logic paths that have larger delays than that allowed by the design specification. The *slack* of a path is defined as the difference between its maximum allowed delay and the minimum actual delay; a path with a negative slack is a *critical* path, and the logic elements and nets along the paths are *critical* elements and nets. The smallest, or *worst*, slack (most negative if negative slacks exist) usually indicates the most critical section of the design, where optimization may result in overall performance improvement. In terms of slacks, the objective of IPR is to eliminate or improve the negative slacks. A *beneficial* optimization improves (increases) the slacks of targeted paths without worsening the worst slacks of the design.

In the physical domain, the delay of a logic path is the sum of the delays of the logic elements and the delays of the nets interconnecting the elements (including the nets that connect logic elements within a hierarchical logic cell, and the nets connecting the hierarchical logic cells using general routing). While logic structure (e.g. fanout size) has impact on the net delay values, it is not used directly in physical delay calculation. Instead, pin-to-pin delays are determined based on actual physical layout.

Delay values of the logic elements and the nets that are inside hierarchical logic cells can be pre-calculated and stored. Acquiring accurate routing delay requires knowing both the placement of the connected cells and the routing paths, since different routing paths may have different delays. In an IPR procedure, such information is available at the beginning, since IPR begins with a completed design with both placement and routing information. On the other hand, our IPR implementation does not include a router, and consequently, the physical optimization part of IPR consists of only placement. Therefore, subsequent physical timing analysis uses a placement based *physical delay calculation* routine to estimate the pin-to-pin routing delay, which approximates the best delay of a route between the given pair of pins.

We have two reasons to use such an approximation. First, in IPR *each* logic transformation requires a physical verification. Including a good quality router will substantially increase runtime. Secondly, we are mostly concerned about the accuracy of the timing information on critical nets, which in most cases will be afforded one of the best (or near best) routes, given that modern FPGA devices in general have ample active routing resources (within a logic utilization limit). Therefore, such an approximation usually does not result in significant loss of accuracy for the nets the concerns us.

In our implementation of IPR we also take several measures to carefully minimize the error in approximation. Firstly, prior to a transformation, we save the timing information that is relevant to the sub-circuit under transformation; if the transformation is rejected, we restore the saved information instead of recalculation, Secondly, we maintain two components of the timing information, the *original* (back annotated from the routed initial design) and the *modification* (calculated based on new or modified placement), separately. The original part remains unchanged;

when a logic element is moved, we only recalculate the *delta* of the delay caused by the movement, so that original timing information can be fully restored if placement change is reversed at any stage of IPR. Finally, the local and incremental nature of IPR also limits the disturbance caused by each round of iteration.

Nevertheless, this is an approximation, and occasionally the inaccuracy can accumulate to a non-neglectable level and affect the quality of the solution. This more likely happens after a large number of IPR iterations, and when the selected sub-circuit has a large number of different critical paths. Overall, however, this is an acceptable compromise.

At the beginning of the IPR procedure, a full chip physical timing analysis is performed. Subsequently, physical timing analysis is performed after each incremental placement; such analysis is carried out incrementally via the update of delay information on modified portion of the circuit (using physical delay calculation, as stated earlier), followed by the update of slacks at the surrounding pins of the sub-circuit, and the propagation of slack changes through related paths.

3.2 Seed Selection

Each iteration of IPR first selects one or more *seeds* which are in general logic elements that have timing violations (i.e. negative slacks). The selection is based on the result of physical timing analysis. If IPR is driven by an interactive user interface, the designer can do the selection according to timing report, or other design information; if an automatic driver is used for IPR, the selection criteria are as follows.

We assign to each critical element a *potential* value representing how likely we can reduce timing violation via improving this and surrounding logic elements. The potential value is a weighted sum of four components for a given critical element. It includes the size of its input sub-netlist of critical combinational logic elements; the placement sparseness of the region where this sub-netlist is placed; its slack value; and the number of registers (and equivalent elements) from which it can be reached and can reach along a critical path. A larger combinational input netlist provides a better chance that changes can be made about the seed via combinational logic optimization, while a sparsely placed neighborhood allows more freedom in exploring alternative logic configurations; these two are *resynthesis potentials*. The worse the slack is, and the more paths the seed is on, the more helpful its improvement will be to overall timing constraint compliance; these are the *timing potentials*. To determine the values of the components in the formula, an analysis of all critical elements is first performed to determine the range of each component, which is used to normalize the value into a range of (0,1]. The weights are configurable per design; for quick improvement, higher weights are given to resynthesis potentials; for better result, higher weights are given to timing potentials.

There are certain structures (shift registers, carry chains, multiplexers) in a design that have been optimized and cannot be further improved. Elements that are part of such structures are excluded.

The potential values are incrementally maintained in a min-max heap [9]. Only a top set is kept; lower potential values are leaked from the bottom of the heap. Automatic selection picks from the top. When optimization happens, the heap is incrementally updated. If improvement is not possible, the next qualified seed will be selected (see below).

3.3 Resynthesis Window

A seed (or a set of seeds) derives a resynthesis window, which the optimization will be focused to. This has two parts, the *logic window*, which is a sub-netlist connected to the seed(s); and a *physical window* where the transformed logic will be placed. The logic window may *grow* if the initial window does not contain enough logic to facilitate improvement; this is either triggered by failed optimization on the existing window, or via demand of a particular optimization operation. The physical window may be *relaxed* to allow extra flexibility and additional resource.

If the seed is a dedicated logic element (e.g. a block RAM), or a register, the logic window will include the seed itself, and the physical window will be the minimum region that encloses the cell in which the seed is placed, as well as the logic cells containing the inputs and outputs of the seed. A logic window so seeded will not grow; the physical window can automatically grow to a prescribed relaxation factor.

If the seed is a distributed RAM, the logic window will include all compatible elements, and their output registers if they exist; the physical window will be the enclosing region of the logic window (with a relaxation factor for growth). A logic window so seeded will not grow.

For a combinational logic seed, the window construction is based on the concept of *fanout-free cone* [4].

Given a network and a node v, a *fanout-free-cone* (*FFC*) of v is a node set containing v and some of its predecessors, such that any path from a member of the set can only reach the outside of the set by exiting from v. We say it is the *FFC of v* (denoted $FFC(v)$), and v is the *root* of $FFC(v)$. The *maximum FFC* (*MFFC*) of v, $MFFC(v)$, is the largest of all $FFC(v)$, which according to [4] is unique and also contains the MFFC of each member.

We can extend these concepts to the case of multiple roots. The *FFC of a set S* of nodes is a node set $FFC(S)$ that contains members of S, as well as some predecessors, such that any path from a member to the outside has to exit from a member of S. The subset of S that can exit the FFC without going through others is the *root set* of $FFC(S)$ and denoted $R(S)$; we can show that $R(S)$ exists and is unique for given S. The *MFFC of S*, $MFFC(S)$, is the maximum $FFC(S)$, and can be shown to be unique for a given S and $MFFC(S) = MFFC(R(S))$. Note that $MFFC(S)$ contains $MFFC(v)$ for each v in S but may contain other nodes as well. Finally, we define the *minimum containing MFFC* of a set S, $MCMFFC(S)$, which *if exists* is the smallest MFFC that *properly* contains S, and has a root set that is disjoint with S. It can be shown that for a given set S, $MCMFFC(S)$ is either non-exist, or is uniquely defined and properly contains $MFFC(S)$ (the proofs are omitted due to space limit). Figure 1 illustrates some of the concepts.

Based on these constructs, for a combinational logic seed the initial logic window includes the logic in the cell (SLICE) containing the seed, as well as the members of its MFFC and the other logic in their cells. The (pre-relaxation) physical window is the region enclosing the hierarchical cells (CLBs) containing these logic elements. Construction based on multiple seeds follows the same rule.

If the window needs to grow, the MCMFFC of the current logic window will be calculated first. If it exists, its root set is used as the new seeds; otherwise, the current logic window is regarded as an FFC, and its root set is used as new seeds. If the new seed set is the same as the old, or if the window size exceeds a pre-set limit, the growth stops, and all seeds that have been involved in the series of growth will be disqualified for further consideration.

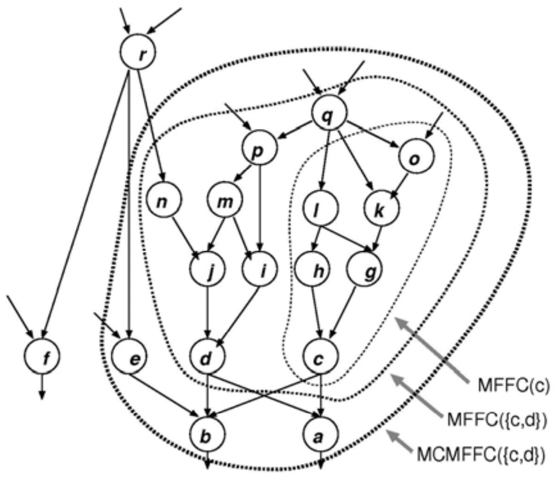

Figure 1. MFFC and MCMFFC

The benefit of using the MFFC structure to determine the resynthesis window is that by definition, MFFCs are re-converged sub-netlists which behave like *super* logic elements, whose internal logic transformations only affect the signals from its root set. This allows maximum flexibility in restructuring the logic and simplifies incremental timing analysis.

3.4 Windowed Optimization

Once a window is determined, it is analyzed for improvement. This involves both logic transformation (netlist change) and physical transformation (placement change). During the process, changes within the window are freely applied as needed; changes outside the window are also allowed but only carried out conservatively. In particular, logic changes to elements outside of the window must not result in new element or the change of the type of an element (that will require a different packing slot); placement changes outside of the window also takes a conservative approach by limiting itself to iterative overlap removal (see the next subsection for details). To ensure consistency of timing data, physical timing information is updated after each placement change, as stated earlier (and detailed in the next subsection).

During the optimization of a window, IPR exploits several types of opportunities and associated optimizations, in the order of the cost to perform the transformations.

The simplest case is that the logic was sub-optimally associated with a (hierarchical) cell and cannot be simply relocated during physical optimization. Via *cell repacking*, we may be able to fix such problems. Another similar case is that the logic in a cell is configured sub-optimally; an alternative configuration, revealed by cell repacking, would reduce its delay.

Similarly, if by modifying certain logic we could reconnect certain critical signals via a faster interconnect, or from a faster equivalent source, via *signal rerouting*, we can achieve immediate delay reduction.

A related, but more complicated case is when a specific type of resource was used, and due to resource distribution on the device, large external interconnect delay was introduced. Via *resource retargeting* we can map the logic onto alternative resources, and use the extra freedom in physical optimization to improve timing.

Finally, the extra delay may be due to a sub-optimally synthesized sub-netlist; *logic restructuring* would explore possible improvements.

Note that these are not mutually exclusive. For example, signal rerouting is helped by limited logic restructuring that generates required signals. Such cross-references are made as needed.

These transformations, in the above order, are considered in the current window, and attempted if applicable,. If one type of optimization is successful but timing violations associated with the current window still exists, the subsequent types of optimizations may also be considered, provided that the preceding types have not made the subsequent ones inapplicable. Each type of optimization is evaluated (via placement and physical timing analysis) and accepted or rejected separately.

It should be noted that these transformations, when computed, are of a tentative nature, since their benefit has to be verified before they are accepted. To make the necessary reverse transformation easy, a temporary hierarchy can be created surrounding the logic window. In so doing, the application and reversal of the transformation becomes toggling between two implementations (the original and the new *views*) of this artificial hierarchy, which is a simple operation. It is important, though, to also properly associate relevant physical and timing information with the two views, and apply and de-apply them accordingly.

3.5 Incremental Placement and Evaluation

Some logic optimization operations in IPR, basically those that transforms one general combinational logic sub-circuit into another one (as opposed to e.g. memory conversion or register movement), may first be evaluated using a simple internal timing model to determine if there is enough potential benefit in the transformation. This model is based on netlist structure as commonly used in technology mapping algorithms [5], but delays of various logic elements and interconnects are varied to reflect the difference caused by their different types. This model is used before and after the logic transformation, to compute a *nominal worst delay* throughout the target sub-circuit; if this becomes worse after transformation, the transformation is rejected without invoking placement.

If a transformation passes this test (or if the test does not apply), physical optimization/evaluation is invoked. As stated earlier, we only perform an incremental placement for this purpose. Therefore, IPR maintains a legal placement throughout the optimization iterations, but routing may not be complete at the intermediate states (after IPR, all signals will be routed). Modern FPGA devices usually have sufficient routing resources for designs with reasonable logic utilization ratios; therefore this is usually not a problem. To minimize the possibility of generating un-routable placement, congestion and utilization control is enforced throughout the placement routines. In all of our experiments, routing can always complete.

The incremental placer consists of three components.

The first component is a quick area placer. It uses a quadratic programming formula to initially place, with possible overlaps, the newly generated/modified logic elements to the best locations based on their peripheral connections. This step is carried out once the logic transformation is applied and old logic elements unplaced from the chip (after this information has been saved). If the new elements form a special packing group, such directive is followed. Since this step is relative to the peripherals, global timing information is not needed. At the end of this step, all new elements have valid placement (but not necessarily legal, due to overlaps), allowing physical delay calculation to be carried out, and incremental physical timing analysis to be completed by adjusting slacks at the peripherals of the transformed sub-circuit, and propagating the slack changes through associated paths.

The second component performs overlap removal, with two possible modes. In a simple mode, it just moves the least preferred element out of an overlapped location to the best vacant location. This is mainly used as an intermediate step. Alternatively, a *competitive displacement* approach is used, where a displaced element may bump another placed element out of its location if the former is deemed more fitting to that location. This can be used as a final placement step in situations where minimum disturbance to existing placement is preferred; for example, when placement disruption outside the physical window is necessary (in which case displaced elements outside of the window may bump placed elements inside the window), or when only a small portion of logic is re-placed. At the end of this step, the design becomes legally placed (or the transformation rejected if legalization fails).

The third component is a timing driven iterative improvement placer which operates mainly in the physical window, but can be relaxed to push some elements outside of the window (in the case of new overlap, they will be re-placed using competitive displacement). This is the most effective, yet timing consuming, portion of placement. While an annealing algorithm may be used, a structural placement is preferred because the trade-off between quality and runtime can be better controlled.

Note that thw two latter placement components only involve movement of already placed logic elements. Consequently, we can keep the timing information up to date throughout the movements, by updating the delays associated with moved elements and their nets, and updating and propagating slack changes, immediately after each movement; this can be done very efficiently. Moreover, by recording the sequence of movement, the movements can be completely reversed and the placement undone (with timing information fully restored as well); therefore, if at a time during placement the transformation is deemed non-beneficial, the original logic/timing/placement can be restored, and the transformation rejected, without side effect.

4. Physical Resynthesis Operations

In this section we will detail and exemplify each type of optimizations used by IPR. As noted earlier, these optimizations are tuned towards FPGA specific architectures. The descriptions in this section apply to the Xilinx Virtex II series FPGA devices. While these *types* of optimizations should be applicable to other modern FPGA devices, certain operations will not be applicable if the specific architectural features they depend on are not available; while other operations can be similarly developed for architectural features not available or used in the following discussion.

4.1 Cell Repacking

This applies to logic ranging from one regular cell (SLICE) up to one higher physical hierarchy (CLB), depending on the type of seeds and the content of the window. We assume that, as the result of the (regular) logic and physical optimizations the precedes IPR, simple swapping of swappable elements among regular cells will not improve timing. We consider the following cases.

For a register seed packed in an IOB, movement to a regular cell within current physical window, between (and including) the cells of its logic inputs and its current location, is explored for timing improvement. Similarly, for a register that exclusively drives an IOB, movement towards the IOB is considered. (A register not directly connected to IOB has been optimized during retiming [18] so is not considered here.)[1]

For a combinational logic seed, its MFFC is analyzed. If the logic fits a pattern that can be implemented using the cascade chain [19] but was not implemented as such, and the required cells can be repacked to vacate the needed resources for the chain without creating worse timing violations, the MFFC is transformed into the form and packed into the cells. An example is shown in Figure 2. This transformation applies only to architectures with a fast cascade chain feature.

This technique only applies to FPGA architectures with fast cascade chains built into logic cells.

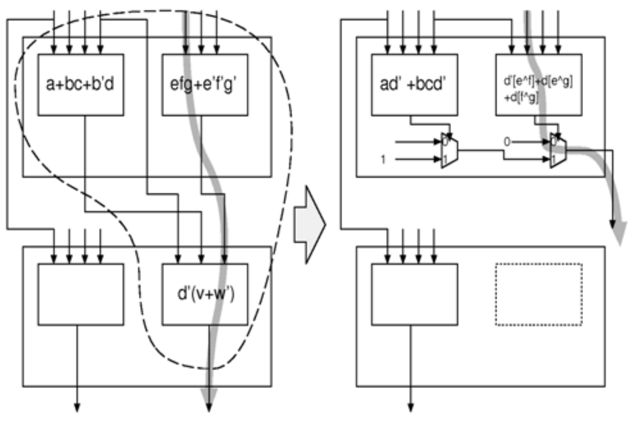

Figure 2. Repacking into cascade chain

4.2 Signal Rerouting

This applies to a LUT seed S with a critical connection that uses external interconnect. We assume that due to the preceding optimizations, it is impossible to convert this into internal signal simply by repacking logic. We consider two cases: if the signal

[1] This is not resynthesis in the strict sense, but is something current physical optimization may not do, since IOB register packing is often done during logic synthesis.

can be internally regenerated, or if a faster external source can be found.

In the first case, we consider the optimization of a critical input if its driver is also a LUT (while other inputs can be driven by anything), by trying to convert *S* into a MUX. In doing so, logic of its input elements may change but their types (LUT, MUX, etc.) can not be changed. If this is possible, we will attempt to convert *S* into a MUX and pack the critical driver into the same cell (again, using iterative overlap removal) to improve timing. (See Figure 3 for example.) If a single critical output of *S* drives a LUT, the same approach also applies.

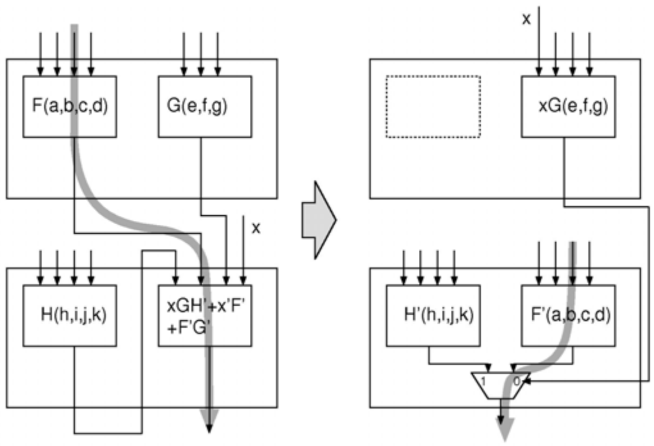

Figure 3. Rerouting via MUX

Moreover, a single critical output may drive a register through its set/reset port, where the register effectively has an OR gate of *S* and its data input *D* attached onto it (with parity, determined by register attributes). If *D* is a LUT, and *S* and *D* share an input *c* that is unate to both, but in different phases, we can reprogram *S* and *D* to exclude c, move the OR function from the register to a MUX controlled by *c*, and use the MUX to drive the data input of the register, thereby converting the critical signal to internal. An example is in Figure 4.

The second case (exploring alternative signal source) applies when the driver *D* of the critical signal is a combinational logic element. We use a precompiled table that maps each logic element to a signature of its primary input set (the non-combinational elements that can reach it via a path of combinational logic elements). First we select each element *L* in the MFFC of the seed *S* that has a better slack than *D*, and has the same primary input set as *D*. Then, we examine if *L* produces an equivalent, or inverted, signal as *D* does; if so, *L* will be a candidate to replace *D*. If that fails, then we examine if a new LUT can be formed using the inputs of *L* to produce such a signal, and can be placed to maintain a better slack than *D* without adding timing violations to others. Once a replacement is found, the signal is rerouted to it. Functional comparisons are done using BDDs and are aborted if intermediate BDD size exceeds a preset limit. This heuristic search of equivalent signal can be improved by an SPFD based approach [7], which is currently under development.

In all cases, the change to logic as well as physical structure is local and minimal.

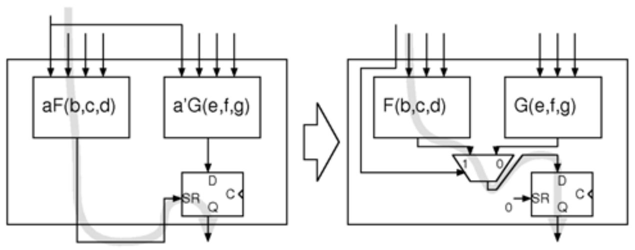

Figure 4. Rerouting for register

4.3 Resource Retargeting

This type of optimization involves the use of dedicated logic resources such as block RAMs and multipliers. Several cases are considered.

First, if a memory element implemented in block RAM has timing violations (that cannot be reduced by placement improvement alone), it is considered for retargeting, according to the following criteria:

If the critical signals are data signals, we will try to split the memory into sections according to the placement of the source and/or destination of the signals, so that each section can be placed closer to those signals. Each section is first targeted for block RAMs. If block RAM resource is not available at designated region and the actual memory size of the section is small, then it will be converted further into distributed RAMs (with necessary auxiliary logic and registers), provided that there are enough resources, and the functionality can be readily implemented using distributed RAMs. An example is shown in Figure 5 (which is part of a real design), where a critical output signal was separated and the block RAM was converted into a mixture of block and distributed RAMs, thereby fixing the timing violation on the output signal (worst output slack is improved from -2.05ns to +2.23ns).

If the critical signal is a read address, we will try to split the memory into two block RAMs using other address lines, and decode this critical read address outside the memory while trying to place the decoding logic near the source of the address, and/or the output destinations.

In other cases, the block RAM will be fully converted into distributed RAMs (plus necessary auxiliary logic and registers) if possible. This transformation is considered only when the actual memory size is small and the physical window is sparsely used.

Similarly, synchronized distributed RAMs can be retargeted to block RAM implementation. For a given seed, the set of compatible RAMs and output registers are in its logic window. These are first grouped into sections based on their proximity to the input and output elements, and combined size; then each section is converted into a block RAM, one block RMA a time, in order of criticality. The converted block RAM(s) are placed and evaluated; if timing is worsened, it is rejected. An accepted block RAM will be subject to further improvement if its most critical signal is a data of read address signal, using the technique described earlier, if possible.

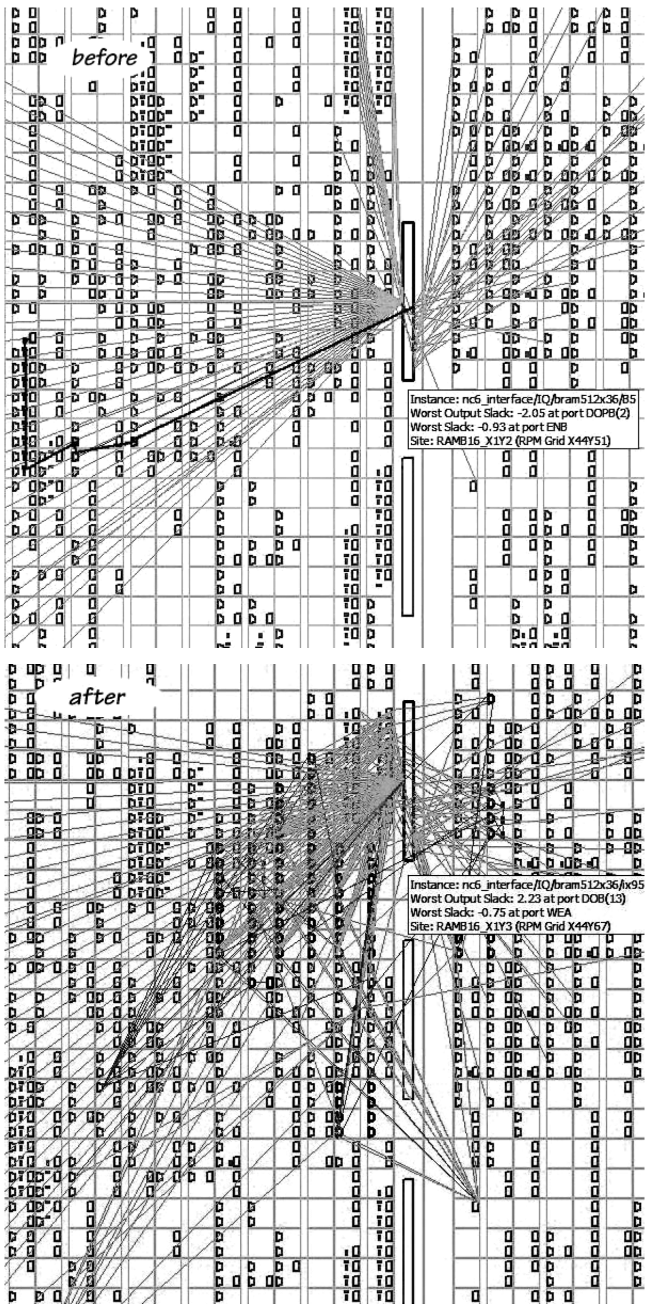
before
Instance: nc6_interface/IQ/bram512x36/B5
Worst Output Slack: -2.05 at port DOPB(2)
Worst Slack: -0.93 at port ENB
Site: RAMB16_X1Y2 (RPM Grid X44Y51)
after
Instance: nc6_interface/IQ/bram512x36/ix95
Worst Output Slack: 2.23 at port DOB(13)
Worst Slack: -0.75 at port WEA
Site: RAMB16_X1Y3 (RPM Grid X44Y67)

Figure 5. Resource retargeting for memory implementation

Finally, for a (grown) window of combinational logic elements, where all seeds are driving compatible registers, and the total input size is within the maximum address width of the block RAM resource, the logic in the window (as well as the output registers for synchronized block RAM) will be tentatively converted to a block ROM [8] to verify timing improvement. When seeds are automatically selected and windows are automatically grown, this is invoked only when a window of combinational logic has grown to its maximum size.

We are currently investigating transformations between block and distributed multipliers.

For different architectures, the rules of conversion may differ.

4.4 Logic Restructuring

This type of optimization applies to a window of combinational logic elements and attempts to reduce the delays from the most critical inputs of the window to a seed, without increasing overall timing violations.

First, a seed with the worst slack is selected, and the sub-netlist containing elements in the window that have the same slacks are constructed and decomposed into MFFCs. Then, the MFFCs are converted into blocks, by selecting in large MFFCs extra elements as block roots, and by duplicating and merging smaller MFFCs into their output blocks, to make the block size at least 4 and at most 8. The decision on promoting an element into a root or demoting one from a root is according to a *ranking* based on the location, type, and output size of an element; an element placed further away from its outputs, a MUX element, and one that has more outputs are favored, as they are more likely to be implemented as roots of logic elements in a efficient optimization solution. Highest/lowest ranked element will be promoted/demoted unless doing so violates the size constraint. After the block partitioning, the elements in each block should have relatively close placement (not counting the duplicated portions). Then, each block is processed, in the order of the distance from the seed. First, the block (which has a single output at its root) is collapsed into a single logic function. Then, we attempt several logic optimizations, in the following order.

- If the block fits a pattern recognized for cascade chain and fits the area constraint (see *cell repacking*), the cascade configuration will be attempted, with an initial placement to the nearest eligible cell to the block root. The result is accepted if the placement can be made feasible, the delay of this block is reduced and the overall timing violation in the window is not increased.[2]
- Delay based functional decomposition [1,3,11] is used to produce a tree of 4-LUTs. Initial placement will keep the root LUT in its current position. If placement is possible, block delay is reduced and the overall timing violation of the window is not worsened, the result is accepted.
- Multi-level Shannon expansion is attempted on all inputs and results are filtered for eligibility according to *cell budget*: for a budget of one/two/four, one/two/three levels of expansions, with final cofactors of no more than 4 inputs, are acceptable since they can fit into as many cells. The acceptable configurations are ranked in order of delay (not counting external routing delay) and tried for placement; the first one that reduces block delay without worsening others is accepted.

If none of the above succeeds, the next block will be processed; if any of the above succeeds, the transformation is committed, and the next block will be processed based on *updated* physical and timing information. This procedure ends when the slack at the seed becomes non-negative, or when all blocks have been processed. If there are other seeds that still have timing violations, the same procedure is repeated on the next worst seed. To prevent unsuccessful blocks from being attempted multiple times, we

[2] This differs from *cell repacking* in that this may cover only a part of the MFFC of a seed, or may cover elements outside of the MFFC.

cache the root and the inputs of such blocks, and skip the processing in subsequent optimizations if the same block is being considered again.

For placement of each block, the competitive displacement placer is used since the tasks are rather small and localized. After all seeds are processed, the timing driven improvement placer is used for further optimization.

If no transformation has taken place during the operation, the current window may grow, either by expanding to the MCMFFC of current window, which introduces new seeds, or by expanding current window to its MFFC, which may add new predecessors; in each case new block structures may be introduced. The cached unsuccessful blocks will be used to avoid the reprocessing of un-improvable parts.

5. Experimental Results

We have implemented a prototyping IPR system based on the approach described in the preceding sections. As stated earlier, the system can be driven via an automatic optimizer that identifies critical sections and applies the appropriate optimizations; or via a user interface, where critical sections are defined by manually selected seeds. We report some preliminary experimental results in this section.

Since our optimization techniques are geared towards real world FPGA architectures and complicated designs, It is difficult to evaluate their effectiveness using simplified device models and MCNC benchmarks. Therefore, we tested our system on a set of real designs from our customers[3], in the setting of real design flow targeting Xilinx Virtex II series FPGAs. Prior to entering IPR, these designs were first compiled and synthesized using the *Precision RTL Synthesis* system [15], placed and routed using the Xilinx *ISE* placement and routing tools [20], then synthesized using the *Precision Physical Synthesis* system [14] which performs retiming, replication, and placement improvement. After IPR, Xilinx *ISE* placement, routing and timing report tools were run again to complete routing and get accurate timing information.

For the automatic flow, we set a limit on the maximum number of resynthesis windows initiated by the driver to 16, and the number of windows for resource retargeting to four[4]. This ensures reasonable run-time, and avoids accumulation of timing inaccuracy due to the approximated routing delays in our physical timing model. In seed selection, we set the weight for timing potentials to be twice as high as resynthesis potentials as such setting is more effective than a few other tested combinations.

Table 1 summarizes the design and device characteristics and pre-IPR/post-IPR status for each of the test cases. Columns 2-5 give the number of logic elements (including LUTs, MUXes and other gates, IO buffers, special logic cells and others) in the design, number of used cells (used SLICEs on the Xilinx Virtex II series FPGA devices), overall utilization (logic/IO), and (rounded) target clock frequency (if multiple clocks exist, the one we were optimizing for is listed). Column 6 shows achieved clock frequency prior to IPR. Columns 7 and 8 show the number of windows IPR attempted on, and number of successful optimizations; Column 9 shows the achieved clock frequency after IPR, and Column 10 the reduction of timing violation. As indicated by the results, these are difficult designs with few places improvable; still, IPR is effective on most of these designs, and on average cut the timing violations by 42.8%, while improving overall frequency by 10.3%.

It should be noted that while theoretically IPR should result in no degradation, the physical timing model we used is not 100% accurate, and errors can accumulate. Therefore, it is possible that the post-IPR performance becomes worse, as shown in a couple of test cases (for which the number of applied transformations is relatively large). We are working on improving the modeling, and identify "good" transformations that only minimally disturb the timing correlation.

With global physical synthesis [14] skipped, the overall IPR results were only slightly better, indicating that IPR and global physical synthesis compliment, rather than compete with, each other.

We have also experimented with the interactive flow, in particular for signal rerouting (a case of truly simultaneous logic and physical transformation) and resource retargeting (a case involves complicated placement), and found it effective. One example is partly shown in Figure 5 of the previous section, where a block RAM is converted into a smaller block RAM and a set of distributed RAMs (the highlighted logic elements in the “after” snapshot, form a sub-netlist that is logically equivalent to the highlighted block RAM in the “before” snapshot) to eliminate the negative slack on a data output. For that design (of ~11,000 logic elements), by applying the optimizations on 3 block RAMs, we were able to completely remove the timing violations.

The run-time for each integrated logic/physical operation is typically within a minute on an engineer PC.

6. Discussions and Future Work

We have presented a new approach to FPGA physical synthesis for timing optimization. Our IPR approach takes advantage of the characteristics of a logically and physical optimized design, and focuses on small, critical sections of the design for timing improvement. This focused nature allows IPR to explore a large spectrum of resynthesis optimizations, and tightly link them to the specific features of modern FPGA devices, to achieve design improvements not reachable by other means.

As designs become larger, and FPGA devices become more versatile, traditional FPGA design flow will have difficulty in achieving timing closure. Physical synthesis will become increasingly important in meeting timing requirements; and tools such as IPR will be essential to allow the designer to claim the last a few megahertz.

We are currently integrating some of the features of IPR into the *Precision Physical Synthesis* system [14] system, while also improving the algorithms. At the same time, we are incorporating some of the techniques proven useful in IPR to traditional RTL synthesis for FPGAs, including a logic synthesis and technology mapping algorithm utilizing cascade chain structure.

[3] These are actually the “difficult” designs we use for quality testing. Some of them are also intentionally over-constrained to push optimization limits.

[4] This does not include iterations due to incremental window growth upon unsuccessful logic restructuring. Also IPR may stop early if it becomes clear that no further improvement is possible, e.g. if the worst path is no longer resynthesizable.

7. REFERENCES

[1] R. Ashenhurst, "The decomposition of switching functions," *Proc. Int. Symp. Theory of Switching Functions*, pp.74-116, Apr. 1957.

[2] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay minimization in lookup-table based FPGA designs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 1, pp. 1-12, January 1994.

[3] J. Cong and Y. Ding, "Beyond The Combinatorial Limit in Depth Minimization For LUT-Based FPGA Designs," *Proc. IEEE International Conf. on Computer-Aided Design*, pp. 634-639, Nov. 1993.

[4] J. Cong and Y. Ding, "On Area/Depth Trade-off in LUT-Based FPGA Technology Mapping," *IEEE Trans. on VLSI Systems*, Vol. 2, No. 2, pp. 137-148, June 1994.

[5] J. Cong and Y. Ding, "Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays," *ACM Trans. on Design Automation of Electronic Systems*, Vol. 1, No. 2, pp. 145-204, Apr. 1996.

[6] J. Cong and Y. Hwang, 'Boolean Matching for LUT-Based Logic Blocks With Applications to Architecture Evaluation and Technology Mapping," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp.1077-1090, Sept. 2001.

[7] J. Cong, Y. Lin and W. Long, "SPFD-Based Global Rewiring," *Proc. ACM/SIGDA International Symp. on Field Programmable Gate Arrays*, pp.77-84, Feb. 2002.

[8] J. Cong and S. Xu, "Technology Mapping for FPGAs with Embedded Memory Blocks," *Proc. ACM International Symp. on Field Programmable Gate Arrays*, pp. 179-188, Feb. 1998.

[9] Y. Ding and M. Weiss, "The relaxed min-max heap," *ACTA Informatica*, Vol.30, pp.215-231, 1993.

[10] R. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," *Proc. 28th ACM/IEEE Design Automation Conf.*, pp.613-619, June 1991.

[11] C. Legl, B. Wurth, and K. Eckl, "An implicit algorithm for support minimization during functional decomposition," *Proc. European. Design and Test Conf.*, pp.412-417, Mar. 1996.

[12] J. Lin, A. Jagannathan and J. Cong, "Placement-Driven Technology Mapping For LUT-Based FPGAs," *ACM/SIGDA International Symp. on Field Programmable Gate Arrays*, pp121-126, Feb. 2003.

[13] R.Murgai, N. Shenoy, R. Brayton, and A. Sangivanni-Vincentelli, "Improved logic synthesis algorithms for table lookup architectures," *Proc. IEEE International Conf. on Computer-Aided Design*, pp.564-567, Nov. 1991

[14] *Precision Physical Synthesis Users Manual*, Mentor Graphics Corporation, 2003

[15] *Precision RTL Synthesis Users Manual*, Mentor Graphics Corporation, 2003

[16] K. Schabas and S. Brown, "Using Logic Duplication to Improve Performance in FPGAs," *ACM/SIGDA International Symp. on Field Programmable Gate Arrays*, pp.136-142, Feb. 2003.

[17] D. Singh, S. Brown, "Integrated Retiming and Placement for Field Programmable Gate Arrays," *ACM/SIGDA International Symp. on Field Programmable Gate Arrays*, pp.67-76, Feb. 2002.

[18] P. Suaris, D. Wang, N. Chou, "Smart Move: A Placement-aware Retiming and Replication Method for Field-Programmable Gate Arrays," *Proc. 5th International. Conf. on ASIC*, Oct. 2003.

[19] *Virtex-II Platform FPGA Handbook*, Xilinx Corporation, 2002

[20] *Xilinx ISE 5 Software Users Manual*, Xilinx Corporation, 2002.

Table 1. IPR Experimental Results

Circuit	#LE	#Cell	Util % (L/IO)	Target (MHz)	Pre-IPR (MHz)	#Win	#Opt	Post-IPR (MHz)	Violation down %
Design1	1941	570	7/82	125	95.63	11	4	109.11	45.9
Design2	1284	465	30/47	70	59.19	7	1	61.77	23.9
Design3	16436	5073	66/37	60	56.34	16	4	59.94	98.4
Design4	3847	1225	79/42	150	133.69	14	1	141.40	47.3
Design5	4691	2131	69/36	70	66.01	11	10	64.90	-27.8
Design6	4555	1472	47/34	133	114.55	9	6	111.74	-15.2
Design7	4588	1760	34/81	90	73.68	16	3	84.06	63.6
Design8	3655	1236	80/67	125	90.21	15	2	106.54	46.9
Design9	2392	982	63/83	133	116.90	6	3	132.87	99.2
Design10	2897	1012	65/60	70	60.42	5	2	77.17	100.0
Design11	4722	1999	65/97	150	113.46	16	11	133.71	54.2
Design12	19270	8334	77/48	50	41.47	16	12	43.12	19.3
Average	**5406**	**2020**		**94.31**	**78.58**	**11**	**4.5**	**86.64**	**42.8**